PANDAS basics

NEW OBJECTS

https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats

import pandas as pd

+ Series: one-dimensional labeled arr.; A series is like a DataFrame with a single column.

+ DataFrame a two-dimensional and labeled array,
Columns store different dtypes

+ Dimensions the number of axes, that are labeled starting at 0.

+ rows **axis 0**; represent the data points, by convention

+ columns **axis 1**; represent the variables, -||-

+ Column names bold names on the top

+ Index labels bold nr's, from 0 to 9 on the left side, for rows only

+ Data everything else inside the cells

+ Cell place set with one row and one column

NEW SERIES

pd.SerieS( ) **input data in a list**
store different datatypes stored as dtype: object
index: additional co, visible on the left, Starts at 0!

```
my_Series = pd.Series([1,'cat','10.2'])
#        0    1
#        1    cat
#        2    10.2
#        dtype: object
```

+ set index parameter, must be given in a **list**

```
ages = pd.Series( [ 20 , 53  ] ,
        index =[ 'John', 'Allen',]);
#        John    20
#        Allen   53
#        dtype: int64
```

+ access to values in a series

- with index number ages[1]
- with index values ages["John"]
- list with multiple values ages[['John', 'Allen']]

NEW DATA FRAME

df.copy( ) obj, with new id( )

pd.DataFrame( ) may take input data in different formats:

[1] dct, with col-names (keys), and col's (values)
!! each list in dct, can have different dtype,

```
pd.DataFrame(
        { "Col_1_Name" : [ Col 1],
         "Col_2_Name"  : [ Col 2] } ,
        index = [ Row_Indexes ]  )
```

[2] list, with list for each row (1 row = 1 emded list
!! items in each list, at correspoiding indexes should have the same dtype (but not necessarly)

```
pd.DataFrame(
        [ [ Row_1 ] , [ Row_2] , .... ] ,
        columns = [ Column_Names ] ,
        index = [ Row_Indexes ]   )
```

[2] data in 2d nnumpy array
!! only one dtype for all data would be allowed

```
pd.DataFrame(
        data = 2D_Numpy_Array,
        columns = [ Column_Names ]
        index = [ Row_Indexes ]  )
```

+ empty df has only col names, indexes and dtype, no data

```
df = pd.DataFrame( index = range( 0 , 2 ), columns = [ 'A' , 'B' ], dtype = 'float')
#              A      B
#      0       NaN    NaN
#      1       NaN    NaN
```

+ df filled with one value - it can be any numeric item, but not a string eg: you may use np.nan because it is a float. pd.DataFrame(np.nan, index=[0,1], columns=['A', "B"])

```
#              A      B
#      0       NaN    NaN
#      1       NaN    NaN
```

IMPORTANT ISSUES

df indexing **for rows;** data points are automatically indexed, starting from 0, but one of the columns in df can be set as new index using **df.set_index("column_name")** method

Copy vs inplace important to know whether a given method introduce changes in a modified obj, or it creates a new object. Most of pandas methods returns a copy, thus you must eaither use inplace = True or df = df.modification() to save changes in an original obj.

Inplace = True ensures that the original object will be modified, not in all methods

.copy( ) ensures that new obj will be created, not in all methods

na_values=["char"] Pandas will automatically recognize and parse common missing data indicators, suchn as NA, empty fields. For other types of missing data use na_
df = pd.read_csv('file_name.csv', na_values=['?'])

EXAMPLES

[1]

```
df = pd.DataFrame( { "col_1":[1, 2, 3],
   "col_2":["item_".join(["",str(x)]) for x in list(range(1,4))],
   "col_3":list(map(chr, range(97, 123)))[0:3] },
   index=["row_".join(["",str(x)]) for x in list(range(1,4))])

# comments:
#        list( map ( chr, range( 97, 123 ) ) ) # GIVES : a,b,..,z
#        ["item_".join(["",str(x)]) for x in list(range(1,4))]
#        generates: item_1,...,ite

#               col_1  col_2  col_3
#        row_1    1      3      3
#        row_2    4      5      6
#        row_3    7      8      9
```

[2]

```
df = pd.DataFrame({    "Car"   :["BMW", "Audi", "VW"],
                      "Price" : [70, 85, 65],
                       "Age" : [2, 2, 4]})
df.set_index("Car", inplace=True)
#        set col, with Car as index
```

LOAD/SAVE & OBJ INSPECTION

LOAD / SAVE CVS

LOAD CSV

pd.read_csv( ) assumes, file has a header, ie. 1st line with col names; eg:
pd.read_csv(" file_name.csv")

header = None; no header
pd.read_csv('file_name.csv', header = None)

names = [....] custom headers, provided in a list
pd.read_csv('file.csv', names = ['Header1' , 'Header2'])

na_values=['string"] set custom values for missing data, in string
pd.read_csv('file.csv', na_values=['?'])

SAVE AS CSV

df.to_csv("File_Name.csv", encoding = 'utf-8', index = False)

"File_Name.csv" can be with path, or else saved in working dir

encoding best use 'utf-8'

index asking to keep row index,
in case you have custom index, its best to keep it.

LOAD OTHER FORMATS

pd.read_excel('file.xls') assumes, file has a header

JSON, HTML, SAS SQL other supported formats

LOAD SEABORN DATASETS

sns.get_dataset_names(); to see all available datasets stored on github
['anscombe', 'attention', 'brain_networks', 'car_crashes', 'diamonds', 'dots', 'exercise', 'flights', 'fmri', 'gammas','iris', 'mpg', 'planets', 'tips', 'titanic']

sns.load_dataset( ) loads given data set as pandas df, using pandas.read_csv
import seaborn as sns
import pandas as pd
titanic = sns.load_dataset('titanic')

+ sns datasets **from:**
https://github.com/mwaskom/seaborn-data

+ titanic dataset **visualizations:**
https://www.kaggle.com/fourbic/visualizing-the-titanic-data-with-seaborn

CLASS & ID

type( ) returns long and confusing class name,

id( ) for object identity, id number

isinstance( ) use no " " around class name! CAUTION: isinstance, returns only one bool valUE (True/False) even when it got a tupple with many classe names. In that case, return True if any class name match to the object.
isinstance(s, pd.Series); isinstance(df, pd.DataFrame)

to_numpy( ) pandas df to numpy array, Important: use copy=True
df.to_numpy(dtype="object", copy=True)

tolist( ) Series to List
df = pd.DataFrame(data=np.arange(1,10).reshape(3,3))
s_list = df.iloc[: , 0].tolist() #. [1, 4, 7]

DIMENSIONS

df.shape eg: df.shape # (3, 2); s.shape # (3, 1) , no brackets!

df.ndim axes number, no brackets!

df.size cells number, no brackets!

DTYPE

df.dtypes returns data type in each col, use no brackets!

df.astype( ) df ['col_1'] = df['col_1'].astype(int)

SEE DATA EXAMPLES (more later)

df.head( ) nr of top rows to display, eg: df.head(5), first 5 rows.

df.tail( ) nr of bottom rows to display, eg: df.tail(5)

df.values returns np.array, no brackets
s.values; df.values

df.unique( ) returns unique values in Series or DF column
df["col_1"].unique()
array(["1" , "2" , "3"] , dtype=object)

DF SUMMARY

df.info( ) returns dtypes, df size etc.. on each col and df.
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
Car 3 non-null object
Price 3 non-null int64
dtypes: int64(2), object(1)
memory usage: 152.0+ bytes

df.describe( ) - **Reruns: count,unique, top, freq, mean, std, min, max, percentiles (25%, 50% and 75%) or count, unique, top, and freq (categorical data)** for each column
- by default **ignores columns with non-numerical data**
- **NaN for column that the stat do no apply** – it is important when using include = "all", you may see NaN is some col's
- **Caution:** df.describe can be run without brackets, but gives less pretty results and some additional text

df.col_1.describe( ) describe( ) applied for only one selected column

include = [ ] what column types to include, these can be: **'all'**, **np.number** (numeric type col), **np.object** (string type columns, here it returns, count, unique, top, and freq for each col), **'category'** (categorical type col)
df.describe(include = [np.number]) # no " "
df.describe(include=['category'])

exclude = [ ] as above with include = [ ], but what to not take
df.describe(exclude=[np.object]) # no obj type col

np.datetime64 Special object from numpy to set date-times eg:

```
s = pd.Series( [    np.datetime64("2000-01-01"),
                    np.datetime64("2010-01-01"),
                    np.datetime64("2010-01-01") ] )
s.describe()
#     count     3
#     unique    2
#     top    2010-01-01 00:00:00
#     req       2
#     first  2000-01-01 00:00:00
#     last   2010-01-01 00:00:00
#     dtype:        object
```

ROWS AND COLUMNS

SEE & CHANGE LABELS

+ SEE ALL +

df.axes see column and row labels, not easy to read.

+ ROW INDEXES +

df.index.values see row names: results in 1d nparray
df.index.values.tolist() # in list

df.index = [ ] name/rename all row indexes, the list with new index names must have the same lenght as the row nr
df.index = list(range(1, 5))

df.rename(index ={}) rename selected row index (dct)
rename (index = {"old row n": "new row n" })
df.rename(index = {"row_1": "bla"}, inplace=True)
# remember about inplace, otherwise its only a view

df.set_index( ) sets one column form df as its index.
Caution: this col is not any longer a column in df.
df.set_index("Col_1", inplace = True)

df.reset_index( ) re-set, to automatic index, starting at 0
df.reset_index(inplace=True)

+ COLUMN NAMES +

df.columns.values returns col names: results in 1d nparray
df.columns.values.tolist() # stored in a list

df.columns = [ ] rename all colnames, the list must be the same lenght, as col number in df; df.columns = ["a", "b", "c"]

df.rename(column ={}) rename selected column names
df.rename (columns = {"old col n": "new col n" })
df.rename(columns={"col_1":"bla"}, inplace=True)
# remember about inplace or copying an entire df.

RE-ORDER & SORT

df..reindex( ) change order of rows/col. MUST Provide all row/col names in a list, but no more. **Caution: it often generates NaN !! see NA later !!**
df.reindex(["row_2", "row_1"] , axis=0); Rows
df.reindex(["col_2" , "col_1"] , axis=1); Columns

sort_index( ) base on index, use ascending = True / False
df.sort_index(ascending = True, axis=0) # ROW
df.sort_index(ascending = False, axis=1) # COLUMN

sort_values( ) sort df col or rows based on values in 1 or >1 of them

[1] Sort Series, based on values in it (can be df col)
df["A"].sort_values( ) # simple sorting

[2] Based on Values s in one df row or col; Caution: If you use column names, you order rows, Thus, axis = 0
df.sort_values("A", axis = 0) # ROWS
df.sort_values(1, axis = 1) # COLUMNS

[3] Based on Values in multiple rows or cols; How?: sorting is primary based on values in col "A", if, "A" has multiple equal val, alg looks for answer in col "B"
df.sort_values(['A' , 'B'] , axis = 0) # ROWS
df.sort_values([1 , 2] , axis=1) # COLUMNS

ADD & REMOVE

df.loc[new_row_nr, :] = [, ,] + ROW,
input data: list with n row items [item 1, item 2, …], where n == col nr. Caution, any row number can be given, if it already exist, you overwrite old data, if it is too large, the index will have weird order

df["new_col_name"] = [, ,] + COL,
input data: list with col n items, where n = rows nr in df. Caution, if you add new list with the same col_name, this col. will be modif.. **It won't create a duplicates, like with concatenate.**

df.assign( ) **+ COL**, eg: add series or numpy array to df

[1] add pandas series to df
s = pd.Series ([45 , 56])
df= df.assign(new_col_name= s.values).copy()
# Key value is not a str, so No " " around

[2] add col by unpacking the dictionary (**)
df.assign(**{"new_col_name" : s.values})
# here a jey should be in " "

drop( ) **REMOVES ROWS OR COLUMNS, returns new obj.**

by labels used to remove rows or col, by their name (label)
df.drop(labels = ["row_name"], axis=0) # - ROWS
df.drop(labels=["col_name"], axis=1) # - COL

by index to remove ROWS, by specifying row index, important: it will remove rows, even when **axis=1**
df.drop(index=[1], axis=0); # drops 2nd row

CONCATENATE

df.concat([list with df's to concatenate])

[1] Concatenate df with different labels at rows or columns
- dfs are joined by corresponding rows and col labels
- if values are missing the function place NaN
- if duplicated, all values will be conserved in new df

df1= pd.DataFrame(np.arange(1,5).reshape(2,2),
columns=["c1","c2"], index=["a","b"]); df1
df2 = pd.DataFrame(np.arange(5,9).reshape(2,2),
columns=["c3","c4"],index=["c","d"]); df2
df.concat([df1, df2]); df

	c1	c2	c3	c4
a	1.0	2.0	NaN	NaN
b	3.0	4.0	NaN	NaN
c	NaN	NaN	5.0	6.0
d	NaN	NaN	7.0	8.0

[2] Concatenate df along given axis (0, row; 1, col), Often creates duplicates !, by default, joined by row
df1 = pd.DataFrame([['a1', 'b1'], ['a2', 'b2']],columns=['A, 'B'])
df2 =pd.DataFrame([['a3', 'b3'], ['a4', 'b4']],columns=['A, 'B'])
pd.concat([df1 , df2], axis=0) # row indexes may be duplicated

	A	B
0	a1	b1
1	a2	b2
0	a3	b3
1	a4	b4

pd.concat([df1, df2], axis=1) # column indexes may be duplicated

	A	B	A	B
0	a1	b1	a3	b3
1	a2	b2	a4	b4

verify_integrity = True pd.concat([df1, df2], verify_integrity=True)
Error: Indexes have overlapping values

DATA EXPLORATION Part 1

EXAMINE THE DATA

+ DATA EXAMPLES

df.head( ) nr of top rows to display, eg: df.head(5), first 5 rows.

df.tail( ) nr of bottom rows to display, eg: df.tail(5)

df.values returns np.array, no brackets
s.values; df.values

df.unique( ) returns unique values in Series or DF column
df["col_1"].unique()

+ BASIC STATISTICS

df.max([axis, skipna, level, …])
df.min([axis, skipna, level, …])
df.mean([axis, skipna, level, …])
df.std([axis, skipna, level, …])
df.sum([axis, skipna, level, …])
df.var([axis, skipna, level, …])
df.round([decimals])
df.sem; standard error

+ RATE OF CHANGE

df.pct_change( ) Returns % change between the current and a prior elements. **Important:** function has two options: **fill NA** (eg: if NA, there is 0% change), and **step size** (how many rows must past before calculating % change).
df.pct_change([periods, fill_method, …])

Eg: 1

```
s = pd.Series([90, 91, 85]); s.pct_change( )
#        0     NaN
#        1   0.011111
#        2  -0.065934
```

Eg: 2

```
df = pd.DataFrame({
...   '2016': [1769950, 30586265],
...   '2015': [1500923, 40912316],
...   '2014': [1371819, 41403351]},
...   index=['GOOG', 'APPL'])
df.pct_change( axis = 'columns' ) # axis to go along
        2016     2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

+ more at (good link)
https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats

MISSING DATA

+ EVALUEATE NaN PROBLEM

df.empty True if df has no data,
ie. no col's, no rows, no brackets after the function!

df.count( ) number of non-NA/null in each column, returns series

df.isnull( ) True if a given cell has NaN
returns df with True/False in each cell
df2.isnull().sum(axis = 0) ; NaN per column
df2.isnull().sum(axis = 1); per row

+ LOCATE NaN

++ In any cell **df.loc[df["A"].isnull( )]**
shows rows with NaN in column A (True)

++ in rows **df[df.isnull( ).sum(axis = 1) > 0]**
True for rows with ≥1 NaN

++ In columns **df.iloc[:,(df.isnull().sum(axis=0)>0).tolist()]**
True for rows with ≥1 NaN

+ REMOVE NaN

df.fillna() replaces all NaN wiht given value
df.fillna(value = 0)

df.dropna( ) Removes row/col with NaN**.** You can control where and how much of NaN is required to remove row/col

df.dropna( )
drop all rows with any NaN (axis=0)

df.dropna(axis = 1)
drop cols with any NaN

df.dropna(how = 'all') # how much of NA
drop a row if it has a NaN in all cols

df2.dropna(subset=['A' , 'B']) # where to find NA
drop a row if it has a NaN in col 'A' or B

+ PROBLEM WITH REINDEXING

df.reindex( ) **function that often generates NaN!**
in eg below, it reindexing is used to re-order, rows (axis=0). However, we gave that function more row_names, than. In the original df. Thus, it adds extra rows, that must be filled with NaN
CAUTION: if i place axis=1, it will erase all data !!!!!

```
df = pd.DataFrame(np.random.randint(10, size=(3, 3)),
        index=['a', 'c', 'e'], columns=['A', 'B', 'C'])
df = df.reindex( ['a', 'b', 'c', 'd', 'e', 'f'], axis=0 ) ; df
#          A     B     C
#    a    3.0   7.0   3.0
#    b    NaN   NaN   NaN
#    c    7.0   6.0   8.0
#    d    NaN   NaN   NaN
#    e    6.0   3.0   4.0
#    f    NaN   NaN   NaN
```

Pandas for data sciernce:
https://www.datacamp.com/community/tutorials/pandas-tutorial-dataframe-python?utm_source=adwords_ppc&utm_campaignid=898687156&utm_adgroupid=48947256715&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=1t1&utm_creative=332602034343&utm_targetid=aud-299261629574:dsa-473406573835&utm_loc_interest_ms=&utm_loc_physical_ms=1003215&gclid=CjwKCAjw5dnmBRACEiwAmMYGORGay3EYGqLcn9N2_rt6xqKuw5VFnsluqtjtNjzTJA6aald85uAd-hoCGZMQAvD_BwE#question7

DATA EXPLORATION Part 2

BOOLEAN SELECTION

+ search for a single value

== <'val'> True if exact match was found, all other False

!= 'val'> False if exact match was found, all other True

df == <'value'> # sometimes, generates an error

df.loc[df.column == <'value'>]

returns, df, with selected rows

+ on specific location in df

df.values == <'val'> # search in all df values, returns, **numpy array** !

df.index==<'val'> # search row indexes;

df.columns==<'val'># search column names

df.col_name==<'val'> search in a given column

df['col_name'] == <'val'>

+ search for multiple values

.isin( ) True if exact pattern is found

.notin( ) False if exact, pattern was found, all other are True

Features of isin() and .notin():

Searches in whole df, rows columns

- Returns series or df with True/False,
- True if any of the values was found

df.isin(["item_1", "item_2"]) ¨

returns boolean df with True for all cells in a df with “item_1” and “item_2”

df.loc[df[“col_name”].isin(["item_1", "item_2"])])]

#. returns a copy of df, with rows, that had “item_1 & item_2” in a given col

df.loc[~df[“col_name”].isin([“value_1”, “value_2”])]

#. -||-, with rows, that DIDN’T HAVE “item_1 &2” in a given col

+ multiple search criteria

& AND

| OR

~ NOT

Comments: Use ( ) parentheses to separate the boolean conditions

df.loc[(df[col_1] == value_1) &
(df[col_2].isin([value_2, value_3]))

returns new df (copy, thanks to .loc), with rows that have value 1 in c ol_1, and value 2 or 3 in col_2

GROUP & SELECT IMPORTANT DATA

select_dtypes Returns, subset of a DataFrame including/excluding columns based on their dtype, more at:
https://www.interviewqs.com/ddi_code_snippets/rows_cols_python

df.groupby( ) Splits the data in a df using arbitrary criteria
Creates a GroubBy object which is a mapping of labels to group names its not a df!

Example: df with cats and dogs
df = pd.DataFrame({
A' : ['dog', 'cat', 'dog', 'cat', 'dog', 'cat', 'dog', 'dog'],
B' : ['one', 'one', 'two', 'three','two', 'two', 'one', 'three'],
C' : np.random.randint(10, size=(8))})

Task: calc. means in C for cats and dogs
means = df.groupby("A")["C"].mean( )
returns means for cats and dogs

Alternative: df.loc[df [' A'] ==' cat' , 'C'].mean( )
Problem: you must do that for each group

df.pivot( ) Allows construction of derivative tables form original df, Takes 3 arguments, that are column names in original df.

index: unique values from that col in original df will become row_names in derivative df.

columns: unique values from that col in original df will become col_names in derivative df.

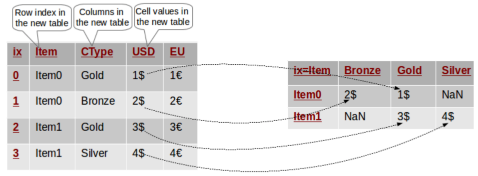
values: values from that column in original df will be used fto fill in cells in new table,

- if none will be found, pivot assign NaN
- if pivot find duplicates, it will show ERROR, ValueError: Index contains duplicate entries, cannot reshape - you must use pivot_tabl

1. Pivot by single value column

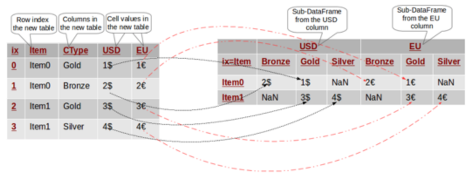
```
from collections import OrderedDict
import pandas as pd
import numpy as np
# collections:, a package with specialized data containers eg: OrderedDict; dct with ordered items, deque; list optimized for inserting & removing items

    table = OrderedDict((
        ("Item", ['Item0', 'Item0', 'Item1', 'Item1']),
        ('CType',['Gold', 'Bronze', 'Gold', 'Silver']),
        ('USD',  ['1$', '2$', '3$', '4$']),
        ('EU',   ['1€', '2€', '3€', '4€'])))
    df = pd.DataFrame(table)
```

df. Pivot(index= ‘items’, columns = ‘CType’, values=’USD’)



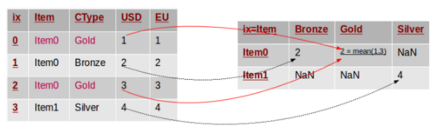
2. Pivot by multiple columns

pdf = df. Pivot(index= ‘items’, columns = ‘CType’)
Pandas will create a hierarchical column index (MultiIndex) that can be accesses in new object: eg:
Pdf.USD.Bronze
print(d[(d.Item=='Item0') & (d.CType=='Gold')].USD.values)
print(p.USD[p.USD.index=='Item0'].Gold.values)



df.pivot_table( ) as pivot(), but it allows dealing with data duplicates, via value aggregation (eg: **aggfunc=np.mean**)

df.pivot_table(index='Item', columns='Product', values='EU', aggfunc=np.mean)



For more see and for original txt on pivoting (figures were taken form there)https://nikgrozev.com/2015/07/01/reshaping-in-pandas-pivot-pivot-table-stack-and-unstack-explained-with-pictures/

PANDAS basics

SLICING

" : " select all

One keyword or Int eg: 1, "row_1", True, "col_2"
- returns only one element, like 1 row or 1 col
- if placed directly in [ ], Returns pdSeries: df.iloc[1]
- when placed in a list: Returns pdDataFrame;

Slice with int. [from : to - 1 : step]
- indexes start at 0 !
- ends not included, eg: 0:4 gives [0, 1,2,3] !!
- Can not be in a list
- step, eg: 0:4:2 gives [0, 2 ,4]

Slice with keywords [from : to : step]
- TO IS INCLUDED!, unlike with int !
- can not be in a list

List [Keywords / exact numbers / True/False]
- eg: ["row_1","row_3"], [1, 3]
- only, labels mentioned in list will be selected
- list can not carry range eg:
  - df.iloc[[0:3:],] # error - List with slice
  - df.iloc[[0, 1, 2], :] # ok. - all selected indexes

SLICING METHODS

df[ ]

Features: indexes and True/False are assumed to be from rows, Key Labels, form columns

df[Slice with Int. / Keywords]
- **rows only**
- df[0:10] # selects row 0-9,
- df[row_1 : row_10] # row:_10 is included

df[List with True/False]; ROWS
- **rows only**
- df[[True, True, False, False, …]] # T/F for each row, if less, only first rows will be taken, and the

df[Column name]
- **columns only**
- single column name, or ≥1 colnames in a list

not accepting
- **Single Numbers**, eg: df[1], (use df[1:2]),
- **Single True/False** df[True],(use df[[True, False, False.,..]]) , or lists with numbers.

df.col_name

df.col_name[ ]
- **column names only**
- return pdSeries
- can be sliced with indexes eg: df.col_1[0:10]

df.loc[ ]

Features; Accepts Key values or True/False only

df.loc[selected_rows]
- **if used for rows,** the selection may be given directly in brackets [], eg: df.loc[row_1:row_10]

df.loc[selected_rows , selected_cols]
- **if used for columns**, you must first select the rows, and then provide values for selecting columns eg: df[:, col_1:col:10], where ":" selects all rows

df.iloc[ ]

Features; accepts the following:
- ranges with numbers (from : to-1 : step)
- numbers in a list
- True/False in a list
- No broadcasting, of true/false

df.iloc[selected_rows]
- df.iloc[1 : 10] # rows 1-9
- df.iloc[[1,2,3,…9],:] # -||-
- df.iloc[[True, True, …], :]

df.iloc[selected_rows , selected_cols]
- df.iloc[:, 1 : 10] # columns 1-9

T/F list is too short List with True/False can be shorter than row or col nr. .loc[] and .iloc[]. do not support broadcasting, Thus, only first rows fitting to that list are selected
- df.loc[:, [True]] # first col only
- df.loc[[True], :] # first row only

Key duplicates eg: rows and columns have the same labels
[ ] returns KeyError!!!

Return Series or df Some selection return pdSeries, other pdDataFrame.
- Series: df.loc[:,"col_1"]
- DF: df.loc[:,["col_1"]]

COPY VS VIEW

Problem 1: Chained Assignment

Error message: `SettingWithCopyWarning

Reasons: Py cannot ensure whether copy or a view was returned

Eg: df[df.col_name == < value >].col_name **# error**

df[df.col_name == < value >] **# ok**

Solution: use .loc(), it always return a copy()

Avoid using chained assigments, like in the above

Problem 2: DF subset is not a copy

Error message: `SettingWithCopyWarning

Reasons: Py doesn't know if it will modify a view or a new obj.

Eg: df2 = df[df.col_name == < value >] # no loc, no copy

df2.loc[0:2, 0:4] ==< value >] # WARNING

Solution: use .copy(), or loc or iloc to create new df

df2 = df[df.col_name == < value >].copy()

Use copy() to ensure no problems (see later)

df = df[df["col_1"] != "valie_1"].copy()

Use .loc operator has several advantages, when using Boolean selection
1. loc always creates a copy()
2. you may use any type of bool and slicing selection together for rows and columns
3. it always require selection or a slice for rows and columns thus, there is no assumptions, like in df[], whether we are ask for row or col

1) df.loc[(df['col_1']== 'value_1') & (df['col_2'] > value_2), :]
2) df.loc[(df['col_1']== 'value_1') & (df['col_2'].isin([value_2]), :]
3) df.loc[(df['col_1']== 'value_1') , 0 : 25]
4) col_sel = [True, False, False, False, True, True, False]
df.loc[(df['col_1'] == 'value_1') , col_sell

SLICING EXAMPLES

SELECT ROWS

[numbers from : to-1 : step, :] or **iloc[[numbers in a list], :]**

```
df[:]  # all rows.
df[0:1] # 1st row only
df.loc[1:2]; error,  you must use df.iloc[1:2]
df.iloc[1] # 1st row, returns pdSeries
df.iloc[ [ 1 ] ] # 1st row, returns pdDataFrame
df.iloc[0:1] # 1st row, DF
df.iloc[0:1,] # - || -
```

[keywords from : to : step , :] or **[[keywords in list], :]**

```
df["row_1"] # error
df["row_1":"row_1"] # 1st row!, correct form!
df["row_1":"row_2"] # 1st and 2nd row !!
df["row_1","row_2"] # error, use df.loc[[]]
df.loc["row_1"] #1st row, shows as pd.series
df.loc["row_1":"row_3"] # 1st to 3rd row
df.loc["row_1", "row_3"] # error # must be in a list
df.loc[ ["row_1", "row_3"] ] # 1 & 3 row only
```

[[list with T/F]]

```
# important, with loc, and iloc  list can be shorter than
row nr, but not with df[ [ ] ],  it must be the same length
df[true], or df.loc[True] # error, T/F must be in a list
df[[True, False, True]] # 1st and 3rd row
df.loc[ [True, False, True] ] #same res.  in all eg. below
df.loc[ [True, False, True], ] # rows == True
df.loc[ [True, False, True], :] # rows == True
df.loc[ [True] ] # to short, returns only the 1st row!
df.iloc[[True]] # 1st row
df.iloc[[True],] # -||-
df.iloc[[True, False, True],] # 1st and 3rd row
df.iloc[[True, False, True],:] # -||-
```

COLUMNS

df.col_name

```
Returns pdSeries; some people like that form
df.col_1 # 1st col!
Caution: col_name can not have spaces
```

iloc[:, numbers from, to-1, step] or **iloc[: , [numbers in a list]]**

```
df.iloc[:,0] # 1st col; Return Series
df.iloc[:, [0]] # -||-; Return DF
df.iloc[:,0:2] # 1st & 2nd column
df.iloc[:, [0,1]] # - ||-
df.loc[,0] # ERROR: add selection for rows
df.loc[,[0:3]] #ERR.: no slices in list, should be [0,1,2]
```

[[keywords in list]] or **[: , [keywords in list]]**

```
df["col_1"] # 1st col!, Retunrns Series
df[["col_1"]] # -||-, Returns DF
df[["col_1","col_3"]] # 1st & 3rd col
df.iloc[“col_1”:”col_3] # ERR. gives df with no rows
df.loc[ "col_1" ] # 1st col, Retunrns Series
df.loc[: ,"col_1" ] # 1st col, Retunrns Series
df.loc[: , [ "col_1" ] ] # 1st col, Returns DF
```

[: [list with T/F]]

```
df.loc[:,[True, False, True]] # col’s 1 & 3, returns  DF
df.loc[:,[True, False]] # returns only 1st col, Caution!
df.loc[,[True, False, True]] #ERR., select rows, eg “:”
df.loc[[True, False, True]] # ERROR, Returns ROWS!
df.loc[True, False, True] # ERROR, must be a list
df.loc[:,[0:2] # ERROR, use iloc instead, for number]
df.iloc[ : , [True]]  # 1st col; Caution!
df.iloc[ : , [True, True, True] ] # 1-3 columns
df.iloc[:,True] # ERROR: TZ/F in a list
```

ROWS & COLUMNS

df.colName[SLICE]

```
Returns pdSeries
df.col_1[0:2] # 1st col, rows 1 and 2!
```

[:, numbers from, to-1, step] or **iloc[: , [numbers in a list]]**

```
df.iloc[0 , ] # row_1, all columns, returns Series
df.iloc[ [ 0 ] , : ] # -||-, returns DF
df.iloc[ [ 0, 1, 2 ] , : ] # row 1-3, all columns
df.iloc[ [ 0 , 2 ] , [ True , False, True ] ]
#              rows 1 & 3, cols 1 & 3
df.iloc[ 0 : 3 : 2 , 0 : 3 : 2 ]
#  -||-, slicing rows and columns with step size =2
```

[: , [keywords in list]]

```
df.loc[ :,"col_1"] # all rows, col_1, Return Series
df.loc[ : , ["col_1"]] # all rows, col_1, Return DF
df.loc[ "row_1", "col_1"] # cell, 1,1
df.loc[ "row_1", [ "col_1", "col_2" ] ]
#              1st row with col 1 and 2
df.loc[ [ True, False, True ] , [ "col_1" , "col_2" ] ]
```

[: [list with T/F]] as shown in example on the above

DataFrame Example

```
import pandas as pd
df = pd.DataFrame(
   { "col_1":[1, 2, 3],
    "col_2":["item_".join(["",str(x)]) for x in list(range(1,4))],
    "col_3":list(map(chr, range(97, 123)))[0:3]
   },
   index=["row_".join(["",str(x)]) for x in list(range(1,4))])
df
list( map ( chr, range( 97, 123 ) ) ) # GIVES : a,b,..,z
["item_".join(["",str(x)]) for x in list(range(1,4))] #item_1,...
```

VALUE MODIFFICATIONS

USEFULL FUNCTIONS

.unique() Finds Unique values in Series or DF column, returns np array; df.age.unique( ) # array([24, 54, 17], dtype=int)

df_1 + df_2
df_1 - df_2 df's or their slices must be of the same size, otherwise it will be added **NaN or inf to resulting df.**

df_1 add(df_2) plus

df_1 sub(df_2]) minus

df_1 div(df_2) division

df_1 mul(df_2) multiplication

+ fill_value used to solve, NA /inf PROBLEM i.e if we add series of different size it will generate NaN or inf (if div by 0)

df_1.add(df_2, fill_value=0)

- added zero on all missing places between two df's!

```
#            col_1   col_2   col_3
#   row_1    0.0     2.0     4.0
#   row_2    6.0     8.0     10.0
#   row_3    0       0       0
```

APPLYING THRESHOLD

df.clip( ) replaces values lower, and/or higher than upper and lower thresholds with these thresholds., see below.

df = pd.DataFrame(np.arange(1,10).reshape(3,3))

df.clip (lower=4, upper=6, inplace=True); df

```
#        0   1   2
#   0    4   4   4
#   1    4   5   6
#   2    6   6   6
```

LAMBDA

Lambda x: function for x

- special expression to store small functions.
- It is advised to use list comprehencion instead in Py 3x

Example; Create a function that turn lower cases into upper cases and apply it to items in df:

capitalizer = lambda x: x.upper( )

df.col_1.apply(capitalizer)

df.col_1.map(capitalizer) # both return the same res.

df.applymap(lambda x: len(str(x))) # nr of characters in each cell.

df.applymap(lambda x: x**2) # works, if all cells are numeric

MAP()

df["column name"].map({ "old_value" : "new_value" })

- only for **Series** or **DataFrame column**
- **Replace ALL existing values** in a series, with new values
- **Introduce NaN** for items from a series that were not in a map dct.
- **Returns a copy**
- Returns error, when applied to df

df = pd.DataFrame(
{'user': [1, 2, 3], 'age': [24, 54, 17], 'sex': ['F', 'F', 'M'] })

df['sex'] = df['sex'].map({ 'F' : 'Female' , 'M' : 'Male' })

Replace F and M in sex column

REPLCACE( )

df.replace("old_value" , "new_value")

df.replace([List of old values", ...] , [List of new_values", ...])

- for **Series, DataFrame, DF Subset**
- like a map(), but **can be used to an entire DF**
- unlike map() **do not exchange unknown elements for NaN!**
- **can replace many values,** stored in two lists at corresponding pos.
- **Returns a copy**

df['sex'] = df['sex'].replace('Female', '1') # replace female to 1

df['age'] = df['age'].replace([24 , 54], ['>20' , '>20']) # many val's

df = df.replace([24 , 3] , ['blabla' , 'test'])

replace many matching values in an entire df

APPLY( )

df.apply(Function_name , axis)

- for **Series, or Axes of DataFrame**
- Used to apply **functions to rows and columns** in df
- **axis = 0** apply goes along axis 0, ie in rows, in each col separately thus, df2.apply(sum, axis=0) provides, sum of each column (new row)

```
↓ |++++++| ↓
↓ |++++++| ↓
↓ |++++++| ↓
  |** ** * *|
```

- **axis = 1** apply performs operations in each row separately going along axis 1
  df2.apply(sum, axis=1);
  # sum from each row

```
 →  →  →
|++++++| * |
|++++++| * |
|++++++| * |
 →  →  →
```

APPLYMAP( )

df.applymap(Function_name)

- for **Entire DataFrame**
- Applying the function **Element-wise**
- **Not a copy in return,** modifications are in place
- **Caution:** may be inefficient with large datasets

```
df = pd.DataFrame(data=np.arange(9).reshape(3,3))
def my_func(x):
        if x <= 4: return 'Small'
        if x > 4: return 'Large'
df.applymap( my_func )
#        0       1       2
#   0    Small   Small   Small
#   1    Small   Small   Large
#   2    Large   Large   Large
```